AFOSR-TR-970168

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
7/8/97

3. REPORT TYPE AND DATES COVERED
9/15/95 - 3/14/97 Final Tech. Report

4. TITLE AND SUBTITLE
SCALABLE DATA PARALLEL ALGORITHMS AND IMPLEMENTATIONS FOR VISION

5. FUNDING NUMBERS
F49620-95-1-0522

6. AUTHOR(S)
R. Nevatia and V.K. Prasanna

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
University of Southern California
Institute for Robotics & Intelligent Systems
Powell Hall 204
Los Angeles, CA 90089-0273

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This effort is about designing, analyzing and implementing scalable and portable parallel solutions to problems in intermediate and high level vision. This is a difficult problem as computations are heterogeneous, symbolic and geometric in nature and use complex data structures such as lists and graphs. Simple data parallel approaches are not sufficient due to uneven distribution of symbolic features among the processors, unbalanced workload, and irregular inter-processor data dependency caused by the input image.

In this work, a realistic model of distributed memory parallel machines which accurately models the features of a parallel machine was proposed. This includes the costs of communication latency, impact of communication patterns on network congestion, available bandwidth and time for synchronization. Using this model, the computation, communication and control characteristics and the memory requirements of the vision algorithms were analyzed. Based on these, an asynchronous parallel algorithm which enhances processor utilization and overlaps communication with computation by maintaining algorithmic threads in each processor was developed. Furthermore, the dynamic task migration technique at an algorithmic level can balance the unpredictable workload in parallelizing intermediate and high level vision problems.

14. SUBJECT TERMS

15. NUMBER IF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

Scalable Data Parallel Algorithms and Implementations for Vision

Final Technical Report

by

Ram Nevatia
Institute for Robotics and Intelligent Systems
University of Southern California
Los Angeles, CA 90089
internet: nevatia@iris.usc.edu

and

Viktor K. Prasanna
Department of EE-Systems, EEB-200C,
University of Southern California
Los Angeles, CA 90089-2562
internet: prasanna@ganges.usc.edu

May 1997

SUMMARY

This effort is about designing, analyzing and implementing scalable and portable parallel solutions to problems in intermediate and high level vision. This is a difficult problem as computations are heterogeneous, symbolic and geometric in nature and use complex data structures such as lists and graphs. Simple data parallel approaches are not sufficient due to uneven distribution of symbolic features among the processors, unbalanced workload, and irregular inter-processor data dependency caused by the input image.

In this work, we propose a realistic model of distributed memory parallel machines which accurately models the features of a parallel machine. This includes the costs of communication latency, impact of communication patterns on network congestion, available bandwidth and time for synchronization. We analyze the computation, communication and control characteristics and the memory requirements of the vision algorithms. Based on these, we propose an asynchronous parallel algorithm which enhances processor utilization and overlaps communication with computation by maintaining algorithmic threads in each processor. Furthermore, our dynamic task migration technique at an algorithmic level can balance the unpredictable workload in parallelizing intermediate and high level vision problems.

To illustrate our ideas, perceptual grouping steps used in an integrated vision system for building detection were used as examples. Our experimental results show that, given 3519 extracted line segments from a $1K \times 1K$ image, both the line and junction grouping steps can be completed in 0.447 seconds on a 32-node SP2 and in 0.390 seconds on a 32-node T3D. For the same grouping steps, a serial implementation required 6.684 seconds and 5.950 seconds on a single node of SP2 and T3D, respectively. The implementations were performed using the Message Passing Interface (MPI) standard and are portable to other High Performance Computing (HPC) platforms.

1 Objectives

This effort is about designing, analyzing and implementing scalable parallel solutions to problems in intermediate and high level vision. Parallel solutions to this problem are characterized by uneven distribution of symbolic features among the processors, unbalanced workload, and irregular inter-processor data dependency caused by the input image. Such problems require development of a sophisticated computational model and techniques. Specifically, the work has the following area of emphases:

1. Design of scalable parallel algorithms and analysis of their performance for a variety of generic geometrical computational problems, such as perceptual grouping, arising in intermediate and high level vision.

2. Developing a generic, realistic model of computation of parallel machines that includes the local memory, communication latency, bandwidth and synchronization overheads, that spans a variety of MIMD architectures and is usable for a variety of symbolic computations.

3. Test the methodology by implementing an integrated vision system that begins with an image and produces high-level descriptions on a versatile MIMD machine such as an IBM SP2 or a Cray T3D. Building detection in aerial images is one of the chosen tasks.

2 Approach

Complexities of the proposed problem preclude the use of massive parallelism based on automatic parallelization or compiler generated mappings. Our approach consists of the following steps:

1. Accurately modeling the features of a parallel machine to develop an abstract coarse grain parallel model of computation that includes the costs of communication latency, impact of communication patterns on network congestion, available bandwidth and time for synchronization.

2. Analyzing the computation, communication and control characteristics and the memory requirements of the vision algorithms.

3. Reorganizing the algorithms to achieve a better match between the characteristics of the computation and the machine. This will typically involve controlling the parallelism at a fine-grain level and letting the processors run asynchronously.

4. Achieving load balancing by dynamic redistribution of the tasks.

In the following sections, we show the result of our approach in parallelizing perceptual grouping steps on an IBM SP2 and a Cray T3D.

3 Research Effort

We have parallelized two perceptual grouping tasks (Line Grouping and Junction Grouping) and implemented it on an IBM SP2 and a Cray T3D. In the following, the definition of this IU task is given. A realistic model of distributed memory parallel machines is presented. Key ideas of the design of a fast algorithm for this task are discussed. The implementation details and experimental results are shown.

3.1 Problem Definition - Perceptual Grouping

We first briefly define the selected IU problem to be solved. In general, vision system operates both in bottom-up (feature extraction, grouping) and top-down fashion (verification). The perceptual grouping process is usually an intermediate-level procedure to group the primitive features detected by low-level processing to form structural hypotheses.

We consider parallelizing the perceptual grouping tasks described in [7]. The grouping procedures are performed in a building detection system for grouping linear features to form structural objects such as rectangles. The rectangles are then used to hypothesize building structures. For the sake of completeness, we briefly outline the processing steps in perceptual grouping. Additional details can be found in [7].

(1) **Line Grouping** - groups line segments which are closely bunched, overlapped, and parallel to each other to form a *line* (a linear structure at a higher granularity level). For each line segment, a search is performed

within the region on both sides of the line segment within a constant width to find other line segments which are parallel to it. The detected segments are grouped to form a *line*.

(2) **Junction Grouping** - groups two close right-angled *lines* to form a *Junction*. For each *line*, a search is performed on both sides of the *line* within a constant width and a fixed size region near its end-points to find *lines* which may jointly form right-angled corner(s). For any two lines which form a "T" junction, the top line will be broken to form two separate lines. To distinguish the linear features detected at different grouping tasks, let *linear* denote the new generated lines and the lines that are not broken.

(3) **Parallel Grouping** - groups two *linears* which are parallel to each other and have "high" percentage of overlap. For each *linear*, a search is performed on a window of size $w \times w$ where w is a given value representing the length of the side of a possible building in the image. We then form a *parallel* by grouping the *linear* with the *linear* found in the window having a difference of slope within a given threshold value and satisfying certain constraints with respect to overlap.

(4) **U-contour Grouping** - forms a *U-contour* if any *parallel* has its two *linears* aligned at one end. A search is performed within the window near the aligned end of each *parallel* to group with *linears* possibly connecting the end-points at the aligned end. If any two *U-contours* share the same *parallel*, a rectangle is formed as a building hypothesis.

To reduce the search time, we store pointers representing image features in an *index array* of the same size as the image. For example, a pointer stored at (x, y) may point to a *junction* feature formed by two segments with (x, y) as their intersection point. Thus, to find a junction near a point of interest only a neighboring area need to be searched.

Let $W(S)$ denote the total size of all the *search windows* generated by a set S of input tokens to a grouping step. The set S represents the collection of token data. These tokens are stored in the index array before each grouping step. The serial time complexity of a grouping step is $O(W(S))$.

3.2 A Model of Distributed Memory Machines

Most state-of-the-art distributed-memory machines can be regarded as a collection of powerful off-the-shelf processors interconnected by a high-bandwidth, low-latency network (see Figure 1). A separate *communication co-processor* at each of the nodes offers the capability to overlap communication with computation. Examples of such machines include Cray T3D/T3E, IBM SP2, Intel Paragon, and Meiko CS2. To exploit such *dual-processor* architectures at an application development level, we must analyze the communication characteristics of such an architecture using a communication library.

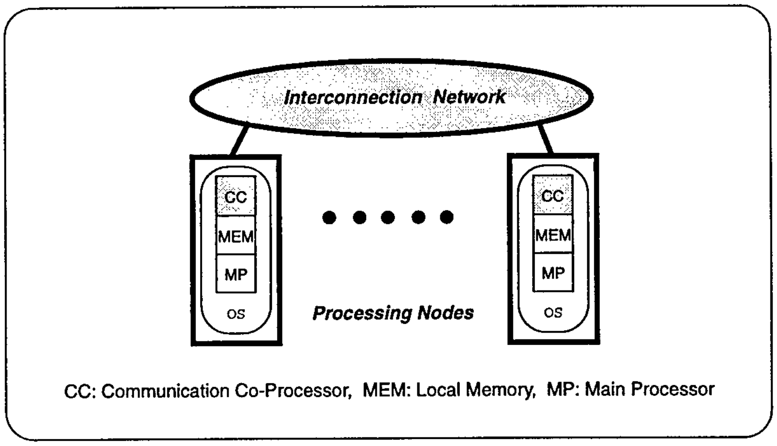

Interconnection Network
CC
MEM
MP
OS
Processing Nodes
CC
MEM
MP
OS
CC: Communication Co-Processor, MEM: Local Memory, MP: Main Processor

Figure 1: An architectural view of coarse-grain MIMD machines

Recently, a standard communication library called MPI has been defined and implemented on many parallel machines [9]. Applications using this library can be ported across platforms with little or no modification to the code. MPI provides two major types of point-to-point communication commands: *blocking* and *non-blocking*. The blocking command does not return to a user program until the communication operation has been completed. The non-blocking command initiates the communication operation, but does not wait for its completion. Therefore, communication can be overlapped with computation using the non-blocking command on machines where the communication operations can be executed by a communication co-processor.

In [4], we measured the communication performance of SP2 and T3D using MPI and then defined our communication model based on those measurements. In this report, we only describe the key parameters of the communication model to capture the overheads in executing parallel algorithms.

Let P denote the number of nodes. Also, let T_b, T_{nb} (μsec) and τ_b, τ_{nb} ($\mu sec/byte$) denote the startup time and the transmission period for data communication using blocking and non-blocking commands, respectively. The startup time is incurred by the software overhead in sending/receiving a message. The transmission period is the time for transferring a unit of data over the network. We make the following assumptions for our analysis:

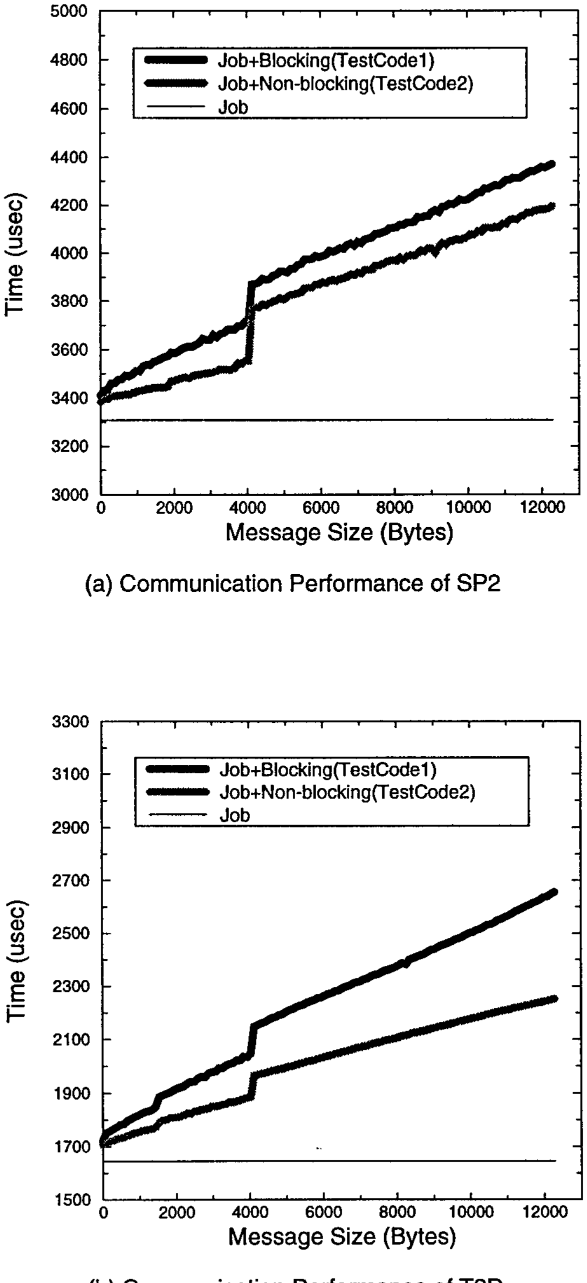
- Sending a message containing m bytes of data takes $T_{nb} + m\tau_{nb}$ time using non-blocking commands. Similarly, it takes $T_b + m\tau_b$ time using blocking commands. These assume that there is no network congestion.
- To perform a collective operation such as broadcast of m bytes of data, $(\log P)$[1]$(T_b + m\tau_b)$ time is required. To perform a barrier synchronization, it takes $(\log P)T_b$ time on machines (such as SP2) which do not have special hardware for barrier synchronization.

The results of our measurements for SP2 and T3D are shown in Figure 2. Note that, to measure the effect of overlapping communication with computation using non-blocking commands, a synthetic job was chosen as the computation step such that its execution time was larger than the round-trip communication time. The data was obtained from a SP2 (Wide66 nodes using MPI-F) at the Maui High Performance Computing Center and a T3D (using EPCC-MPI) at the Pittsburgh Supercomputing Center.

From these results, the startup time and the transmission period can be computed using a linear least-squares fit. The communication times using blocking and non-blocking commands were computed first from the measured data. Then, a line fitting was performed on the observed communication times using messages of length up to 4 $KBytes$ on SP2 and up to 64 $KBytes$ on T3D. The communication parameters resulting from a line fitting are summarized in Figure 3. Due to non-linearity in the line fitting, the values of T_{nb} and T_b may be slightly different from the time to send a zero-length message.

The ratio of startup time to transmission period is in the range of several thousands. To reduce the communication time, the high startup time as well as the message size must be considered. For communicating many short messages between a pair of processors, messages can be combined into a

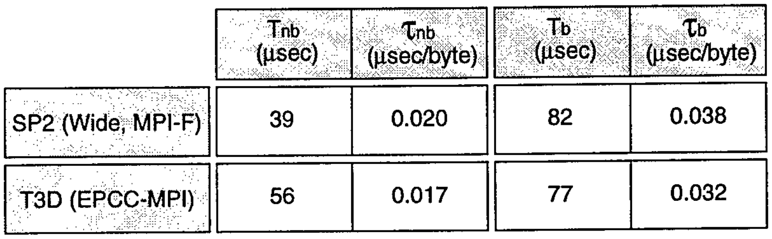
[1] All logarithms in this report are to base 2.



(a) Communication Performance of SP2

(b) Communication Performance of T3D

Figure 2: Communication performance of SP2 and T3D using various sizes of messages



	T_{nb} (µsec)	τ_{nb} (µsec/byte)	T_b (µsec)	τ_b (µsec/byte)
SP2 (Wide, MPI-F)	39	0.020	82	0.038
T3D (EPCC-MPI)	56	0.017	77	0.032

Figure 3: Communication parameters using MPI on SP2 and T3D

longer message to be transmitted as a single unit. We call this *message packing*. Also, the values of T_{nb} and τ_{nb} are affected by the structure of the user program. If the user program provides enough computation to fully overlap communication, T_{nb} and τ_{nb} have the values shown in Figure 3. Otherwise, their values approach T_b and τ_b as the amount of computation is reduced. To exploit the machine features of communication co-processors, the computation needs to be reorganized such that the communication can be fully overlapped.

3.3 Parallel Algorithms

In this section, we develop parallel algorithms for the perceptual grouping task described in Section 3.1. Three parallel algorithms are discussed: *scatter-gather*, *task partitioning*, and *task migration* algorithms. The scatter-gather algorithm is developed for reducing the communication cost caused by irregular inter-processor data dependency. On the other hand, the task partitioning and task migration algorithms are developed for distributing unbalanced workload. In these two algorithms, problems caused by the irregular inter-processor data dependency are simply avoided by the use of an all-to-all broadcast operation. In the following, only the line grouping step is discussed in detail. The same approach can be applied to the other grouping steps. Also, processor, processing node, and node are used interchangeably in the following sections.

A. A Scatter-Gather Algorithm

Let P and $n \times n$ denote the number of processors and the image size, respectively. The input to each processor is an image block of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$, along with the line segments extracted from the image block during the pre-processing phase (i.e., linear feature extraction). Let set S denote the collection of the extracted line segments. The output is a set of lines formed. *Linear feature extraction* can be parallelized by partitioning the $n \times n$ image array into P blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ and then performing the contour detection and linear approximation operations for a partitioned image block in each processor. A parallel implementation of the linear feature extraction has been reported in [3, 4].

Following the sequential algorithm for the line grouping step [7], line segments which are closely bunched, overlapped, and parallel to each other are grouped to form a line. For each line segment, a search is performed on both sides of it (within a 4-pixels width region) to find other line segments parallel to it and subsume them if they are shorter. In the example shown in Figure 4, shorter segment c is grouped into longer segment d to form a new line. Following terminology is used in this section:

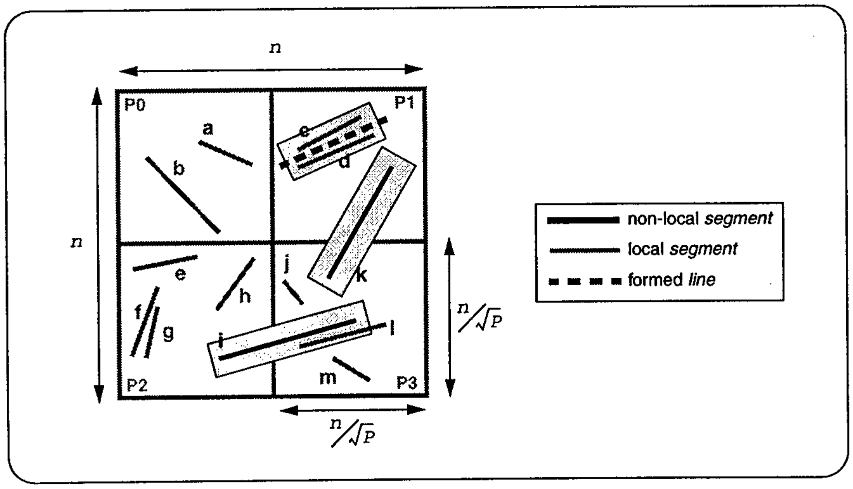


Figure 4: An example illustrating the notation using 4 processors

- *token*: a data structure containing information about an input line segment, such as endpoint coordinates, length, orientation, and average contrast of the line segment. The total number of input tokens is $|S|$. In the following, line segments, segments, and tokens are used interchangeably. For the sake of explanation, a token performing a search is denoted as a *source* token, whereas a token to be examined in the search window is denoted as a *target* token. For a source token s, the search window generated by it is denoted as $\mathrm{W}(s)$.

- *Owner*(s): a processor which stores a source token s. In the example shown in Figure 4, the owner of source tokens e, f, g, h, i is processor P_2, and the owner of j, k, l, m is P_3.

- *Domain*(P_i): an image block of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ assigned to a processor P_i during the pre-processing phase. In the example shown in Figure 4, Domain(P_2) is the lower left block of size $\frac{n}{2} \times \frac{n}{2}$. We assume that each processor has information about the domain of the other processors as well as that for itself. For instance, each processor has information about Domain(P_0), Domain(P_1), Domain(P_2), and Domain(P_3).

- *Remote*(W(s)): the set of processors whose domain overlaps the search window W(s), i.e., $\mathrm{Remote(W}(s)) = \{P_i | \mathrm{Domain}(P_i) \cap \mathrm{W}(s) \neq \emptyset, P_i \neq \mathrm{Owner}(s)\}$. In the example shown in Figure 4, $\mathrm{Remote(W}(i)) = \{P_3\}$.

- *local, non-local* tokens: a token s having empty (non-empty) Remote (W(s)) is denoted as a *local* (*non-local*) token.

Grouping of local line segments at a processor can be performed independently of other processors as there is no data dependency between the processors. In the case of non-local line segments, however, target tokens located within the search windows must be collected from other processors. After the collecting the target tokens, the grouping of these tokens can proceed.

The inter-processor data dependency can be managed in a *scatter-and-gather* fashion. In the scatter phase, each processor sends a source token s having non-empty Remote(W(s)) to Remote(W(s)) processors. Each processor generates search requests for each non-local source tokens. In the gather phase, each processor P_i sends back the target tokens located in

```
Procedure: A Scatter-Gather Algorithm for Line Grouping
Step 1: Create Index Array to store local segments.
Step 2: Generate search requests for non-local segments.
Step 3: Scatter the search requests.
Step 4: Perform window search for each request.
Step 5: Gather line segments located in the requested area.
Step 6: Store the gathered line segments into the Index Array.
Step 7: Perform grouping operations.
```

Figure 5: Outline of the scatter-gather algorithm. The code is executed in each processor.

W(s)∩Domain(P_i) to Owner(s). Note that, each processor generates target tokens for each source token received in the scatter phase. In the example shown in Figure 4, P_2 sends a source token i to Remote(W(i)) = P_3 in the scatter phase. P_3 sends back a target token l to Owner(W(i)) = P_2 in the gather phase. To reduce the startup time (defined in Section 3.2) in both phases, the source and target tokens destined to the same processor are combined into a single message during each phase. This algorithm is denoted as *scatter-gather* algorithm and is shown in Figure 5.

Note that the communication during the execution of the grouping step can be modeled as many-to-many personalized communication with *message length variance* (as there will be as many as P messages, typically of different lengths). Such message length variance can be reduced by smoothing out the outgoing traffic variance at the processors. For simplicity, however, we do not address this issue in this report. Techniques to handle this problem can be found in [10].

Indeed, the scatter-gather algorithm can be regarded as a parallel solution that employs an "owner compute rule": the grouping operation for each source token is executed by a processor initially storing it (i.e., the owner of the token), once the problems caused by the inter-processor data dependency are avoided by the scatter and gather phases. The disadvantage of this algorithm is that the workload may be distributed unevenly among the processors. The workload of a processor depends on the input data. Furthermore, processors perform additional computation to generate the search requests and target tokens for non-local source tokens. A possible solution to distribute the workload and avoid the extra computations is to let each processor broadcast all its tokens to all the other processors. Once a processor receives the broadcast tokens from all the other processors, it can perform the grouping operations without any inter-processor data dependency. However, the workload of the processors should be distributed carefully since it depends on the input data in a non-trivial way; the size and shape of the search window is different for each symbolic data. We describe two techniques to distribute the workload in the following sections.

B. A Task Partitioning Algorithm

A load balancing technique which performs a task partitioning step at the beginning of the computation is described in this section. An additional technique which migrates the initially partitioned tasks during the computation is described in the next section.

Following terminology is defined to describe the two load balancing techniques:

- *task*: the computational work to be performed for a source token.
- *workload* of a task: the time to execute a task.

For simplicity, the input tokens are assumed to be already replicated at each processor by performing an all-to-all broadcast operation. The objective of load balancing is to assign the tasks to the processors such that total workload is distributed evenly across the processors.

A possible technique for balancing the workload of the processors is to estimate the workload of the tasks and then using it to distribute the workload across the processors at the beginning of the computation. The workload of each task can be estimated by computing the size of the search window. Then, the list of tasks (size $|S|$) is partitioned such that the workload assigned to each processor is nearly the same (i.e., $\frac{W(S)}{P}$). This technique is similar to other approaches [1, 2, 5, 6] in that all the processors participate in the initial load balancing step.

To investigate the effect of task partitioning and that of task migration separately, we denote a parallel line grouping algorithm using the above task

```
Procedure: A Task Partitioning Algorithm for Line Grouping
Step 1: Perform all-to-all broadcast operation.
Step 2: Create Index Array to store all line segments.
Step 3: Partition the list of tasks.
Step 4: Perform grouping operations for the assigned tasks.
```

Figure 6: Outline of the task partitioning algorithm. The code is executed in each processor.

partitioning as the *task partitioning* algorithm. Although this algorithm can distribute the total size of all the search windows evenly across the processors at run time, the actual workload depends on the shape of the search window and the number of tokens within the window in a non-trivial way. Therefore, an additional step to re-distribute the unbalanced workload on-the-fly is needed.

C. A Task Migration Algorithm

To re-distribute the unbalanced workload after the initial task partitioning step, task migration from a heavily loaded processor to a lightly loaded processor needs to be considered. To perform the task migration efficiently, however, several issues should be addressed.

(1) **Classification of Processors**: The first issue is to determine whether a processor is in a suitable state to participate in a task migration. In general, task migration is performed between a heavily loaded processor and a lightly loaded processor. In parallelizing the perceptual grouping steps, however, classification of a processor into a heavily/moderately/lightly-loaded category can change over time (due to the nature of the workload which may evolve differently from a prior workload estimate). Task migration based on such a heavily/moderately/lightly-loaded classification could cause unnecessary and repeated migrations. For example, if a task migration is performed from a heavily loaded processor to a lightly loaded processor, the transfer of the task may cause the lightly loaded processor to become a heavily loaded processor. This necessitates transfer of that task to another processor, and

this *processor thrashing* may repeat indefinitely.

Instead of using a classification that varies with time, we choose a *busy-idle* classification of processors. In this strategy, a task migration is performed only when a processor becomes idle by the processor that becomes idle. Thus, the processor thrashing problem can be avoided. Let *taker* denote an idle processor taking over a task initially assigned to another processor. Similarly, let *giver* denote a busy processor transferring one of its tasks to a taker. Since a taker in our strategy is otherwise idle and there is only one taker per giver, any taken-over task is guaranteed to be performed by the taker.

(2) **Unit of Migration**: The second issue is to determine the unit of a task migration. Since communicating a message is expensive in message passing machines, the overhead in transferring task(s) needs to be considered. Also, different number of tasks assigned to each processor makes programming difficult, particularly in checking for termination condition.

To manage these problems, we define a flexible notion of a group of tasks assigned to a processor and call this a *supertask*. In our strategy, the unit of a task migration is a supertask rather than a task. A supertask is non-preemptable since transferring a partially executed supertask is expensive. Tasks are grouped such that all the processors hold the same number of supertasks initially. This simplifies programming. The number of supertasks can be determined by a programmer considering the number of processors and the total number of tasks. For instance, the tasks initially assigned to a processor can be grouped into three supertasks as follows: (a) a *lower* supertask consists of tasks such that the sum of the areas searched by these tasks is nearly equal to $0.2 \times \frac{W(S)}{P}$, (b) a *middle* supertask consists of tasks such that the sum of the areas searched by these tasks is nearly equal to $0.6 \times \frac{W(S)}{P}$ and (c) an *upper* supertask consists of tasks such that the sum of the areas searched by these tasks is nearly equal to $0.2 \times \frac{W(S)}{P}$ (see Figure 7(a)).

(3) **Processor Pair for Task Migration**: The third issue is to decide the possible partner for a task migration. For simplicity, we restrict possible partners of a processor to its neighbor processors. However, our strategy can be extended easily to non-neighbor partners. As in the task partitioning algorithm, the list of tasks is partitioned into P blocks and then each block is assigned to a processor. For each processor, we denote a *lower* and an *upper* neighbor as the processor having the preceding and the subsequent block

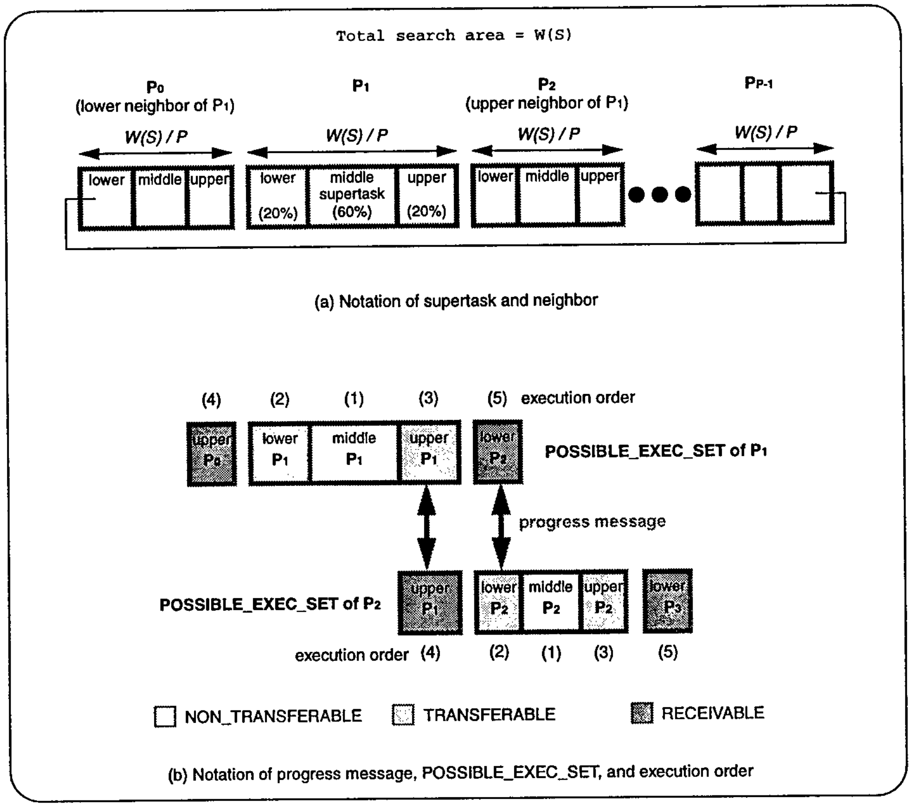


Figure 7: An example illustrating the notation

of tasks, respectively. The list of tasks is considered as a circular list: P_0 becomes the upper neighbor for P_{P-1}, and P_{P-1} becomes the lower neighbor for P_0 (see Figure 7(a)).

(4) **The State of a Partner**: The fourth issue is how to recognize the state of the neighbor processors. A possible solution is to let an idle processor poll its neighbor processors. In the message-passing programming style, however, collecting status information of neighbor processors requires two-way communication. Furthermore, in order to provide a quick response to the polling messages, processors should continuously check for the unpredictable arrival of such messages. This continuous message checking causes additional overhead. To reduce this continuous checking overhead, we let each processor

disseminate its state to its neighbor processors before executing a supertask. Since each processor has the complete list of tasks, each processor sends the index of a completed supertask to its neighbor processors. We denote such a message as a *progress message.* By checking these messages, each processor can recognize the current state (i.e., the remaining workload) of its neighbor processors.

(5) **Tasks to Migrate**: The final issue is to decide which supertask is to be migrated from a giver to a taker. In general, once the pair of processors for a task migration is decided, the giver sends the task(s) to be migrated to the taker. This strategy causes overhead in transferring the supertask. To reduce the communication overhead, we again exploit the facts that each processor has the complete list of tasks and additional tasks are not created dynamically during the computation.

For each processor, we first determine three sets of supertasks as follows:

- NON_TRANSFERABLE: supertasks which were initially assigned to the processor and must be executed by the processor.
- TRANSFERABLE: supertasks which were initially assigned to the processor and can be taken over by its neighbor processors.
- RECEIVABLE: supertasks which were initially assigned to its neighbor processors and can be taken over by the processor.

Note that, the above 3 sets are pairwise disjoint. For the sake of explanation, we denote the union of the above 3 sets as the POSSIBLE_EXEC_SET. For the example shown in Figure 7(b), the set NON_TRANSFERABLE for P_1 consists of a middle supertask initially assigned to it. The set TRANSFERABLE for P_1 consists of a lower supertask and an upper supertask which were initially assigned to it; whereas the set RECEIVABLE for P_1 consists of a lower supertask initially assigned to P_0 and an upper supertask initially assigned to P_2.

We also assign an execution order to the supertasks in the set POSSIBLE_EXEC_SET. In Figure 7(b), a supertask having a lower assigned number in the execution order is executed earlier than all supertasks with higher numbers. For instance, P_1 checks for a progress message destined to its RECEIVABLE supertask from P_2 (shown as "lower P_2" in Figure 7(b)) after executing the 4 supertasks according to the execution order. If the progress message has not arrived yet (i.e., P_2 did not start executing the supertask), P_1 sends

```
Procedure: A Task Migration Algorithm for Line Grouping
Step 1: Perform all-to-all broadcast operation.
Step 2: Create Index Array to store all line segments.
Step 3: Post non-blocking receives for "done/takeover" msgs.
Step 4: Partition the list of tasks.
Step 5: Send "done" msg and perform grouping operations for assigned tasks.
Step 6: If initially assigned tasks are completed, go to Step 8.
Step 7: Check termination condition. If FALSE (no "takeover" msg), go to Step 5.
Step 8: Check termination condition. If FALSE (no "done" msg), go to Step 9.
Step 9: Send "takeover" msg and perform grouping operations for taken-over tasks
        which were initially assigned to neighbors in Step 4. Go to Step 8.
```

Figure 8: Outline of the task migration algorithm. The code is executed in each processor.

a progress message and executes the supertask. Later, P_2 will check for a progress message destined to its TRANSFERABLE supertask to P_1 (shown as "lower P_2" in Figure 7(b)) after executing its *middle* supertask. At this time, P_2 can recognize that P_1 already executed both its TRANSFERABLE supertask to P_1 and its RECEIVABLE supertask from P_1. Note that, the net effect of this scenario is that a task migration is performed; P_1 executes the *lower* supertask initially assigned to P_2.

The main advantage of our strategy is that each processor can execute the supertasks in the set POSSIBLE_EXEC_SET by following the pre-established execution order. Before executing any supertask either in the set TRANSFERABLE or in the set RECEIVABLE, each processor only needs to check for the progress messages to determine whether the supertask has already been executed by its neighbors. Consequently, each processor can terminate itself without the need for barrier synchronization. For the example shown in Figure 7(b), P_1 terminates locally after all the supertasks in its POSSIBLE_EXEC_SET are executed either by itself or by its neighbors. We use MPI non-blocking commands (MPI_Isend, MPI_Irecv, MPI_Test) to communicate *asynchronously.*

A parallel line grouping algorithm using the above task migration technique, in addition to the initial task partitioning step, is denoted as the *task migration* algorithm (see Figure 8).

This task migration technique can be regarded as an emulation of *reactive systems* at an algorithmic level. Unlike other algorithmic techniques [1, 2, 5, 6] for balancing workload, our technique manages load balancing as an ongoing reactive process in the sense that the evolving workload continually forces reconsideration of a pre-established plan.

3.4 Analysis of Time Complexity

Let P and S denote the number of processors and the set of line segments in an input image. The $n \times n$ image is assumed to be partitioned into P blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ during the pre-processing phase (linear feature extraction). The communication time of each algorithm will be analyzed using the communication model defined in Section 3.2. Since the number of operations performed in the grouping steps depends on the distribution of the input image, we define the following notation:

Let d denote the density of the line segments in an $n \times n$ input image. In Section 3.1, we defined W(S) as the total size of all the search windows generated by a set S of input line segments. For the sake of explanation, d is defined as the ratio of W(S) to the image size. The range of d is assumed to be $0 \leq d \leq 1$.

Let $W(S)_{max}$ denote $max(W(S)_i \mid 0 \leq i < P)$ and $W(S)_{avg}$ denote $(\sum_{0 \leq i < P} W(S)_i)/P$, where $W(S)_i$ represents the total size of search windows examined by a processor P_i. Then, $L_{estimate}$, defined as $\frac{W(S)_{max} - W(S)_{avg}}{W(S)_{avg}}$, denotes the degree of unbalance in the estimated workload caused by the distribution of the line segments in the input image. Note that $\frac{W(S)_{max}}{W(S)_{avg}}$ reaches its maximum when all the line segments are concentrated in a single processor. In this case, $W(S)_{avg} = \frac{W(S)_{max}}{P}$ leading to $L_{estimate} = P - 1$. If the line segments are distributed evenly across the processors, $L_{estimate}$ becomes 0. Thus, the range of $L_{estimate}$ is $0 \leq L_{estimate} \leq P - 1$.

In the sequential algorithm, the number of operations performed is $C_1|S| + C_2W(S)$, where C_1 is the average number of operations performed in the indexing step and C_2 is the average number of operations performed in the searching step. Since $W(S) \gg |S|$ and $W(S) = n^2d$, the sequential algorithm can be completed in $O(n^2)$ time.

A. Scatter-Gather Algorithm

In analyzing the execution time of the scatter-gather algorithm, the distribution of the line segments over the partitioned image blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ needs to be considered. Since the number of source tokens initially stored in a processor is upper bounded by $\frac{n^2}{P}$, Step 1 in Figure 5 can be completed in $O(\frac{n^2}{P})$ time. In Step 2, the execution time depends on the total number of search requests generated in a processor. The number of non-local segments in a processor is bounded by $4 \times \frac{n}{2\sqrt{P}}$. It is because the line segments do not overlap and the number of such line segments is less than half the number of boundary pixels in a partitioned image block (see Figure 4). If each non-local line segment has a search window that crosses over $2\sqrt{P}$ processors, then the line segment generates as many as $2\sqrt{P}$ requests. Thus, the total number of requests generated in a processor is upper bounded by $\frac{2n}{\sqrt{P}} \times 2\sqrt{P} = 4n$. Therefore, the time to compute Step 2 is bounded by $O(n)$.

The communication time in Step 3 is determined by the number of *packed* messages and the total lengths of the messages generated in a processor. The number of packed messages and the total lengths of the messages are upper bounded by $O(P)$ and $O(n)$, respectively. Thus, Step 3 can be completed in $O(PT_b + n\tau_b)$ communication time using blocking commands (T_b, τ_b are defined in Section 3.2). The time to perform Step 4 is determined by the number of source tokens received by a processor. Each processor can receive at most $4 \times \frac{n}{2\sqrt{P}}$ search requests since the number of search requests received is bounded by the number of image block boundary pixels. Since the area searched for each request is bounded by $O(\frac{n}{\sqrt{P}})$, Step 4 can be completed in $O(\frac{n^2}{P})$ time.

Similar to Step 3, the communication time in Step 5 depends on the number of *packed* messages and the total length of the messages generated in a processor. The number of messages and the total length of the messages are upper bounded by $O(P)$ and $O(\frac{n^2}{P})$, respectively. Thus, Step 3 can be completed in $O(PT_b + \frac{n^2}{P}\tau_b)$ communication time using blocking commands. The time to perform Step 6 is determined by the number of target tokens received by a processor. Since the number of target tokens collected for each source token is upper bounded by $O(\frac{n}{\sqrt{P}})$, Step 6 can be completed in $O(\frac{n^2}{\sqrt{P}})$

time.

The time to perform Step 7 can be classified into the time spent on local segments and the time spent on non-local segments. The grouping operations for local segments can be completed in $O(\frac{n^2}{P})$ time. The area searched for each non-local segment is bounded by $O(n)$, and at most $\frac{2n}{\sqrt{P}}$ such segments can be stored in a processor. Therefore, the grouping operations for non-local segments can be completed in $O(\frac{n^2}{\sqrt{P}})$ time.

Theorem 1 *Given a set of line segments extracted from an $n \times n$ image, line grouping can be performed in $O(\frac{n^2}{\sqrt{P}})$ computation time and in $O(PT_b + \frac{n^2}{P}\tau_b)$ communication time using the scatter-gather algorithm on a platform having P processors, $1 \leq P \leq n^2$.*

B. Task Partitioning Algorithm

Compared with the scatter-gather algorithm, there is only one communication step (Step 1 in Figure 6) in the task partitioning algorithm. If each processor generates m' bytes of message, then the broadcast operation will be completed in $O([P-1][T_b + m'\tau_b])$ communication time using blocking commands. Since $Pm' = O(n^2)$, the communication time can be represented as $O(PT_b + n^2\tau_b)$.

Steps 2 and 3 can be completed in $O(|S|)$ time, and Step 4 requires $O(\frac{W(S)}{P})$ time. If $W(S) \geq P|S|$, the total computation time becomes $O(\frac{W(S)}{P})$. Since $W(S) = n^2 d$, the computation can be completed in $O(\frac{n^2}{P})$ time. For $P \leq 256$, the constraint $W(S) \geq P|S|$ can be satisfied as the average window size $\frac{W(S)}{|S|}$ is usually in the range of a few hundreds in performing a grouping step.

Theorem 2 *Given a set of line segments extracted from an $n \times n$ image, line grouping can be performed in $O(\frac{n^2}{P})$ computation time and in $O(PT_b + n^2\tau_b)$ communication time using the task partitioning algorithm on a platform having P processors, $1 \leq P \leq \frac{W(S)}{|S|}$.*

C. Task Migration Algorithm

In the task migration algorithm, the overheads to maintain a fixed number of supertasks (Steps 3, 6-8 in Figure 8) can be completed in $O(1)$ computation time and $O(T_{nb})$ communication time using non-blocking commands.

Therefore, the worst case complexity of the computation time of the task migration algorithm is the same as that of the task partitioning algorithm. The inter-processor communication can be completed in $O(PT_b+T_{nb}+n^2\tau_{nb})$ time.

Theorem 3 *Given a set of line segments extracted from an $n \times n$ image, line grouping can be performed in $O(\frac{n^2}{P})$ computation time and in $O(PT_b + T_{nb} + n^2\tau_b)$ communication time using the task partitioning algorithm on a platform having P processors, $1 \leq P \leq \frac{W(S)}{|S|}$.*

In practice, our task scheduling based on the *progress messages* can speed-up the computation by migrating tasks from a busy processor to an idle neighbor. Performance improvement by using the task migration algorithm is discussed in the next section.

3.5 Parallel Implementation

Our algorithms were implemented on the SP2 (employing Wide nodes) at the Maui High Performance Computing Center and on the T3D at the Pittsburgh Supercomputing Center. 1, 4, 8, 16, and 32 nodes were used. The experiments were conducted in a dedicated mode to accurately measure the time. A set of extracted line segments detected from an 1024 × 1024 Modelboard image was used. The raw image, the extracted line segments, and the results of the line and junction grouping steps on a 32-node T3D are shown in Figures 9-10.

The code was written using C and MPI message passing library. The total length of the code is around 27,000 lines. In the following, the reported time is the minimum time observed over 10 executions of the same code on an input image.

To show the effect of the number of supertasks in the task migration algorithm, two sets of experiments were conducted using three supertasks and nine supertasks (see Figure 11).

The "takeover" and "done" messages were implemented as null messages with different "tag" fields. Note that, due to the non-zero communication time for the "done" message to transit from a node to another node, there is a possibility of two processors executing the same supertask. To minimize such duplication, the taker checked for the "done" message after completing

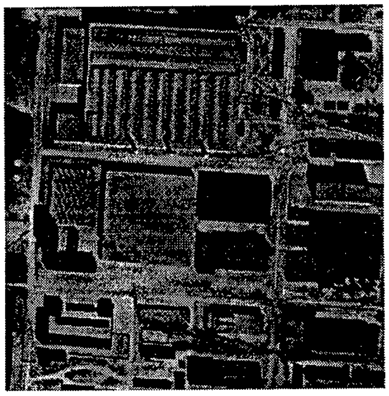
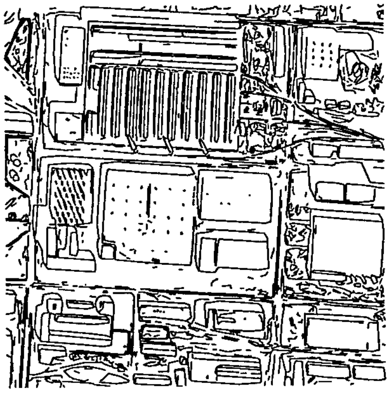


Figure 9: An 1024 × 1024 Modelboard image.



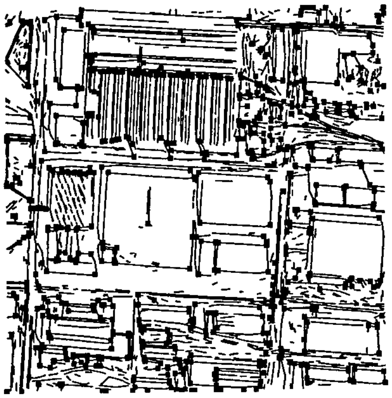
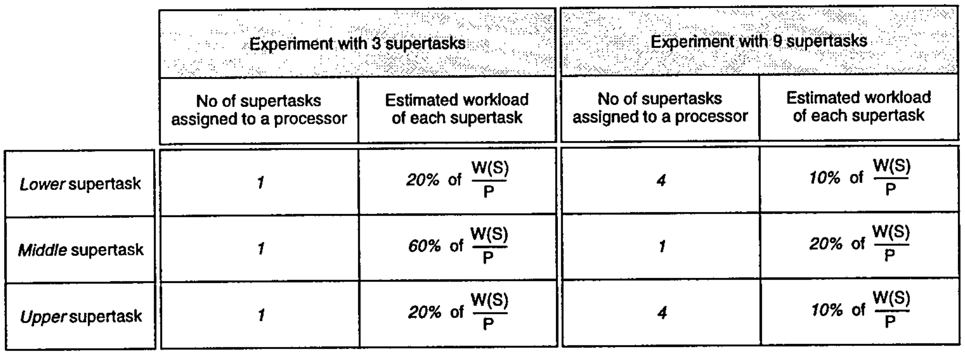


Figure 10: Extracted line segments from the 1024 × 1024 Modelboard image (left) and results of the line and junction grouping steps (right). The detected junctions are shown as small black squares.



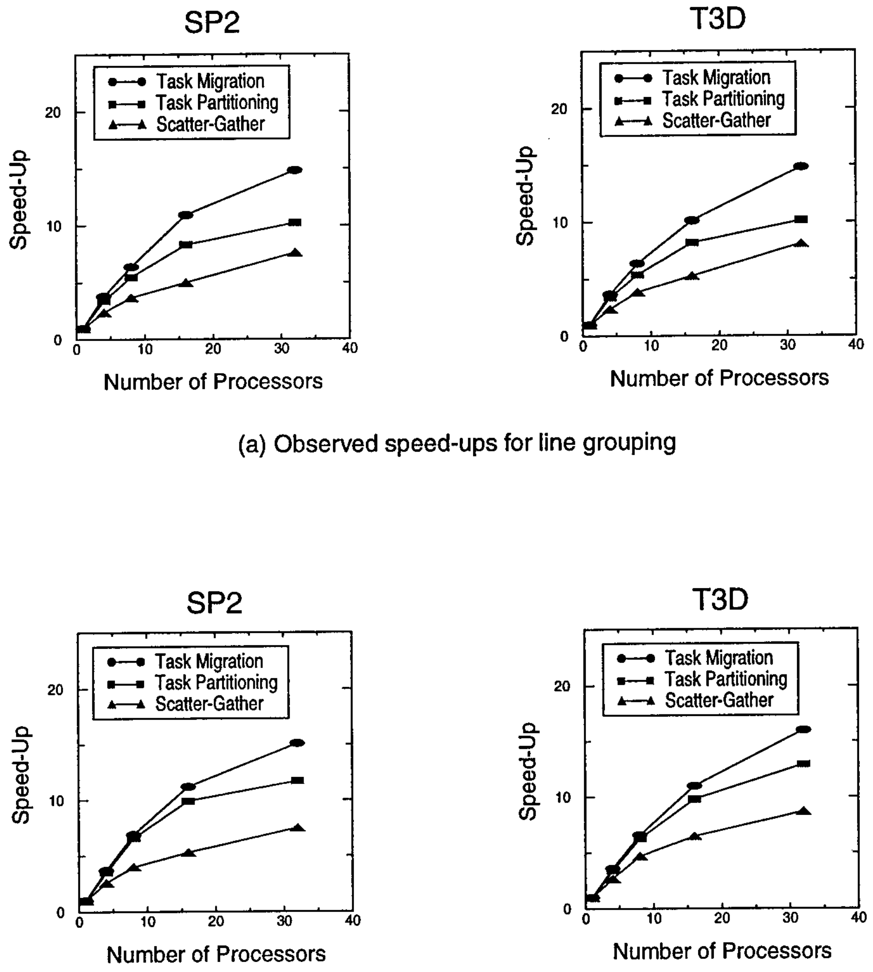
	Experiment with 3 supertasks		Experiment with 9 supertasks	
	No of supertasks assigned to a processor	Estimated workload of each supertask	No of supertasks assigned to a processor	Estimated workload of each supertask
Lower supertask	*1*	*20%* of $\frac{W(S)}{P}$	*4*	*10%* of $\frac{W(S)}{P}$
Middle supertask	*1*	*60%* of $\frac{W(S)}{P}$	*1*	*20%* of $\frac{W(S)}{P}$
Upper supertask	*1*	*20%* of $\frac{W(S)}{P}$	*4*	*10%* of $\frac{W(S)}{P}$

Figure 11: Comparison between the two sets of experiments to show the effect of the number of supertasks in the task migration algorithm.

each task and quit if such a message had been received. Since any taker would otherwise stay idle, total execution time does not increase.

The observed speed-ups of the scatter-gather, the task partitioning, and the task migration algorithm with 9 supertasks for line and junction grouping steps are shown in Figure 12. Given 3519 extracted line segments (detected from an $1K \times 1K$ image), the task migration algorithm completed both the line and junction grouping steps in 0.447 seconds on a 32-node SP2 and in 0.390 seconds on a 32-node T3D. For the same input, the execution time of the task partitioning algorithm was 0.621 seconds on a 32-node SP2 and 0.540 seconds on a 32-node T3D; while the execution time of the scatter-gather algorithm was 0.884 seconds on a 32-node SP2 and 0.713 seconds on a 32-node T3D. A serial implementation of the same grouping steps required 6.684 seconds and 5.950 seconds on a single node of SP2 and T3D, respectively. The total execution times of the line and junction grouping steps using the task migration algorithm with 9 supertasks are shown in Figure 13.

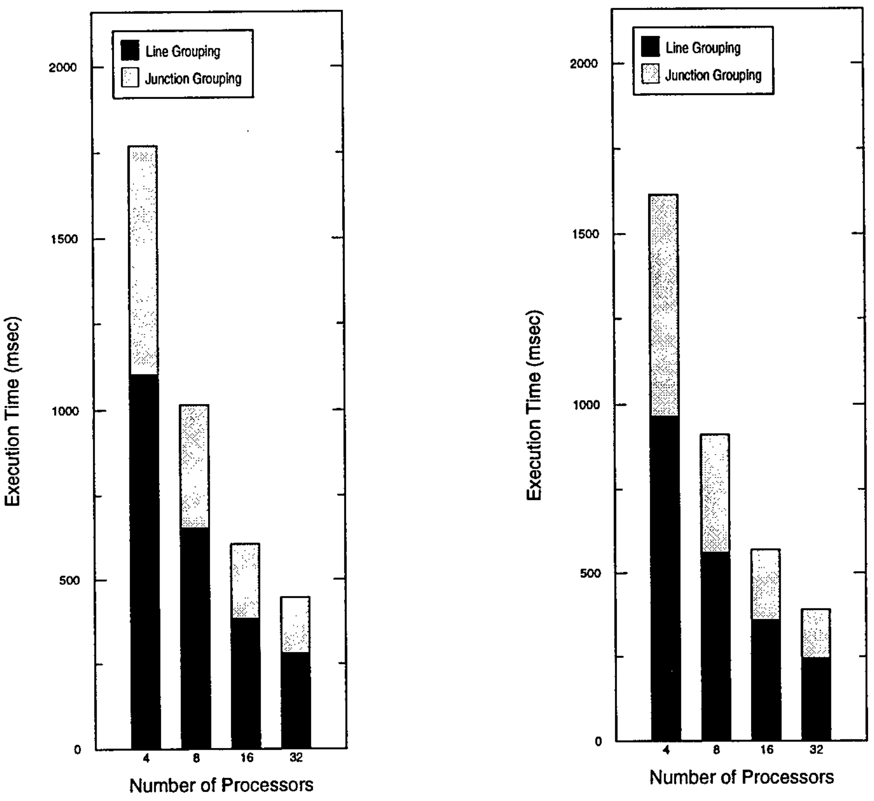
Although we have shown the experimental results on SP2 and T3D, our code written using MPI library can be easily ported to other parallel machines (e.g., Intel Paragon and Meiko CS2, among others).



(a) Observed speed-ups for line grouping

(b) Observed speed-ups for junction grouping

Figure 12: Observed speed-ups for the the line and junction grouping steps.



(a) Total execution time on SP2

(b) Total execution time on T3D

Figure 13: Total execution time for the line and junction grouping steps.

3.6 Conclusions

Computer vision has significantly different computational characteristics compared with other Grand Challenge problems in scientific and numerical computations. Parallelizing such vision computations is characterized by uneven distribution of symbolic features among the processors, unbalanced workload, and irregular inter-processor data dependency caused by the input image. Compiler-generated mappings have been partially successful in solving regularly structured problems and have been useful in applications where compiler-induced "inefficiencies" are hidden by "GFLOPS" need of the computation. Such an approach is highly limited by the availability of compiler techniques and does not seem to be suitable for irregularly structured problems in computer vision.

In this work, we designed scalable data parallel algorithms for a variety of generic vision problems arising in intermediate and high level vision. Also, the algorithms were analyzed by developing a realistic model of computation of parallel machines. Finally, the developed methodology was tested by parallelizing an integrated vision system. The experimental results are very encouraging and show the promise of being extended to realize integrated vision systems which can support interactive performance.

Although the details of our parallel algorithms are application specific, they utilize techniques that are of general interest. Thus, these developed techniques can be used for solving other challenging vision computations. Various future research avenues emerge from this effort:

- Characteristics of vision computations depend on the content of input data as well as the size of the input data. Thus, it is difficult to predict the performance of parallel algorithms for irregularly structured vision computations. In addition to asymptotic analysis of performance, a more accurate methodology for predicting performance for a given input data using a specific number of processors is needed.

- So far, the study of parallel algorithms for vision has investigated individual vision tasks. For an integrated vision system, providing a parallel solution to the complete system requires integrating parallel solutions for individual components. This requires the study of overheads involved in combining various solutions and fine tuning the existing parallel algorithms such that the overheads incurred due to data conversion between various tasks are minimized.

4 Selected Publications

1. Yongwha Chung, Viktor K. Prasanna, and Choli Wang, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP2," IEEE Workshop on Computer Architectures for Machine Perception, pp. 294-301, September 1995.

2. Choli Wang, Viktor K. Prasanna, and Youngwon Lim, "Parallelization of Perceptual Grouping on Message-Passing Machines," IEEE Workshop on Computer Architectures for Machine Perception, pp. 323-330, September 1995.

3. Choli Wang, Viktor K. Prasanna, and Yongwha Chung, "Parallel Implementation of Perceptual Grouping Tasks on Distributed Memory Machines," Image Understanding Workshop, pp. 905-911, February 1996.

4. Yongwha Chung, Viktor K. Prasanna, and Choli Wang, "Parallel Algorithm for Linear Approximation on Distributed Memory Machines," Image Understanding Workshop, pp. 1465-1472, February 1996.

5. Yongwha Chung and Viktor K. Prasanna, "An Asynchronous Parallel Algorithm for Symbolic Grouping Operations in Vision," Euro-Par'96, pp. 123-130, August 1996.

6. Yongwha Chung and Viktor K. Prasanna, "Image Feature Extraction on Coarse-Grain Parallel Machines," submitted to IEEE Transactions on Pattern Analysis and Machine Perception, September 1996.

7. Yongwha Chung, Jongwook Woo, Ram Nevatia, and Viktor K. Prasanna, "Load Balancing Strategies for Symbolic Vision Computations," International Conference on High Performance Computing, pp. 263-269, December 1996.

8. Yongwha Chung, Choli Wang, and Viktor K. Prasanna, "Parallel Algorithms for Perceptual Grouping on Distributed-Memory Machines,"

submitted to Journal of Parallel and Distributed Computing, March 1997.

5 Professional Personnel

Ram Nevatia	Co-principal Investigator
Viktor K. Prasanna	Co-principal Investigator
Yongwha Chung	Student Research Assistant
Chochin Lin	Student Research Assistant
Choli Wang	Student Research Assistant
Jongwook Woo	Student Research Assistant

6 Interactions

We presented two papers titled "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP2" and "Parallelization of Perceptual Grouping on Message-Passing Machines", at the IEEE Workshop on Computer Architectures for Machine Perception in Como, Italy, September 1995. The paper "An Asynchronous Parallel Algorithm for Symbolic Grouping Operations in Vision" was presented in Euro-Par'96, Lyon, France, August 1996. The poster "A High Performance Vision System for Photo Interpretation" was presented at the Supercomputing Conference in Pittsburgh, USA, November 1996. And, the paper "Load Balancing Strategies for Symbolic Vision Computations" was presented at the International Conference on High Performance Computing in Trivandrum, India, December 1996. We also submitted the paper "Parallel Algorithms for Perceptual Gropuing on Distributed-Memory Machines" to the Journal of Parallel and Distributed Computing, March 1997.

References

[1] D. Bader and J. Já Já, "Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection", *Proc. of International Parallel Processing Symposium*, pp. 292-301, 1996.

[2] A. Choudhary, B. Narahari, and R. Krishnamurti, "An Efficient Heuristic Scheme for Dynamic Remapping of Parallel Computations", *Parallel Computing,* Vol. 19, pp. 621-632, 1993.

[3] Y. Chung, V. Prasanna, and C. Wang, "A Fast Asynchronous Algorithm for Linear Feature Extraction on IBM SP2", *Proc. of Computer Architectures for Machine Perception,* pp. 294-301, 1995.

[4] Y. Chung and V. Prasanna, "Image Feature Extraction on Coarse-grain Parallel Machines", submitted to *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1996.

[5] I. Foster and B. Toonen, "Load Balancing Algorithms for the Parallel Community Climate Model", *Technical Report, Argonne National Laboratory*, 1995.

[6] D. Gerogiannis and S. Orphanoudakis, "Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, pp. 994-1013, 1993.

[7] A. Huertas, C. Lin, and R. Nevatia, "Detection of Buildings from Monocular Views of Aerial Scenes using Perceptual Grouping and Shadows", *Proc. of Image Understanding Workshop*, pp. 253-260, 1993.

[8] D. Lowe, *Perceptual Organization and Visual Recognition*, Kluwer Academic Press, MA, 1985.

[9] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, 1994.

[10] C. Wang, V. Prasanna, and Y. Lim, "Parallelization of Perceptual Grouping on Distributed Memory Machines", *Proc. of Computer Architectures for Machine Perception*, pp. 323-330, 1995.